접미사 배열에서의 패턴 검색 알고리즘

최용욱[O] 박근수
서울대학교 전기,컴퓨터공학부
{ywchoi[O], kpark}@theory.snu.ac.kr

Pattern Search Algorithm in Suffix Arrays

Yong Wook Choi[0] Kunsoo Park
School of Computer Science & Engineering, Seoul National University

요 약

접미사 배열은 긴 문자열에 대한 효율적인 패턴 검색을 위해 널리 쓰이는 자료 구조로서 지금까지 접미사 배열을 이용하여 텍스트 T 안에서 패턴 P를 검색하는 O(|P|·|Σ|), O(|P|·log|Σ|)시간 알고리즘(|Σ|:알파벳 크기)들이 발표되었다. 본 논문에서는 O(|P|)시간 알고리즘을 제시하고, 기존의 알고리즘들과 비교한 실험 결과를 보여준다.

1. 서론

1.1. 접미사 배열

접미사 배열[1]은 접미사 트리[2,3]와 함께 대표적인 패턴 검색을 위한 인덱스 자료 구조이다. 특히 최근 생물정보학에서의 매우 긴 길이의 염기 서열이나 웹 상의 방대한 양의 문서들에 대한 패턴 검색이 필요해짐에 따라 저장 공간을 적게 사용하는 접미사 배열을 이용한 검색이 주목을 받게 되었다.

1.2. 접미사 배열에서의 패턴 검색 알고리즘

길이가 m인 패턴 P가 길이가 n인 텍스트 T 내에 존재하는가를 결정하는 문제를 푸는 방법으로 Manber와 Myers[1]는 추가적인 테이블을 이용하여 최대 O(m+ logn) 시간이 걸리는 알고리즘을 제시하였으며, 최근에는 상수 알파벳 상에서 최대 O(m)시간이 걸리는 알고리즘들이 발표되었다[4,5]. 그러나 크기가 |Σ|인 알파벳을 고려할 경우, Abouelhoda-Ohlebusch-Kurtz 알고리즘[4]은 O(m·|Σ|)시간, Sim-Kim-Park-Park 알고리즘[5]은 O(m·log|Σ|)시간이 걸리게 된다. 본 논문에서는 최대 O(m)시간인 알고리즘을 제시하고 앞의 두 알고리즘과 비교 분석한 후, 실험 결과를 보여준다.

2. 알고리즘

2.1. 알고리즘의 개요

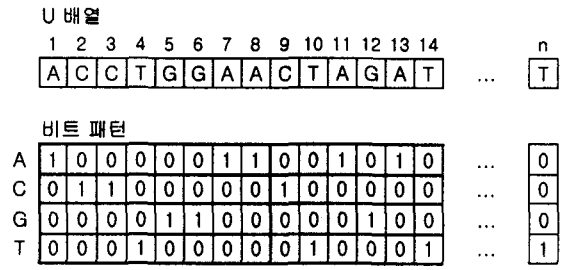
본 논문에서 제시하는 패턴 검색 알고리즘의 기본적인 방법은 패턴의 마지막 문자부터 시작하여 앞으로 한 문자씩 이동하며 해당되는 패턴을 찾아나가는 방법으로 Sim-Kim-Park-Park 알고리즘과 동일하다. 그리고 이러한 검색 방법을 위해 사용한 가상의 배열인 U, V 배열은 Ferragina와 Manzini[6]에 의해 개발되었으며, Sim-Kim-Park-Park 알고리즘에서도 사용되고 있다.

2.2. 알고리즘의 구현

P[j..m]의 패턴의 위치를 찾은 후, 그 결과를 이용하여 P[j-1..m]의 패턴의 위치를 찾기 위해서는 다음과 같이 정의되는 N(i, σ)값을 계산하는 과정이 필요하다.

N(i, σ) = U 배열에서 1부터 i까지에 있는 σ의 개수
(i:U 배열 상의 인덱스, σ:문자)

이 값을 상수 시간에 얻을 수 있도록 구현하는 방법은 다음과 같다.



[그림 1] U 배열로부터 비트패턴 생성

먼저 [그림1]과 같이 각 문자별로 U 배열 상에 그 문자가 있는 위치는 '1'을, 없는 위치는 '0'을 기록하여 비트 패턴을 만든다. 그 후, 다시 각 문자별로 서브테이블I과 서브테이블II를 만든다. 서브테이블I을 만들기 위해 각 문자의 비트 패턴을 가상의 큰 구간($\log^2 n \approx 1024$)으로 나눈 후, 처음부터 각 구간이 끝나는 지점까지의 '1'의 누적 개수를 서브테이블I에 차례로 저장한다. 서브테이블II에는 위의 큰 구간을 각각 가상의 작은 구간($\log n \approx 32$)으로 다시 나누어 위와 마찬가지로 큰 구간 안에서의 '1'의 누적 개수를 저장한다. 마지막으로 작은 구간의 절반의 비트($\log n/2 \approx 16$) 로 표현할 수 있는 모든 수에 대해 각 수를 인덱스로 하여 각 수의 이진수 표현이 가지는 '1'의 값을 저장하는 Lookup 테이블을 만든다.

2.3. N(i, σ)값을 상수 시간에 얻는 방법

주어진 σ에 대해 σ의 서브테이블I을 이용하여 i번째 위치가 속한 큰 구간의 앞 구간까지의 '1'의 누적 개수 N_1를 구한다. 다음으로 σ의 서브테이블II를 이용하여 i번째 위치가 속한 큰 구간 내에서의 직전 작은 구간까지의 '1'의 누적 개수 N_2를 구한다. 마지막으로 Lookup 테이블을 이용하여 i번째 위치가 속한 작은 구간 내에서의 i번째까지의 '1'의 누적 개수 N_3을 구한다. 결국 $N_1+N_2+N_3$가 N(i, σ)값이 된다.

3. 메모리 사용 공간 분석

위 알고리즘은 n에 비례하는 알파벳 크기 만큼의 배열이 필요하므로 메모리 사용 공간의 복잡도가 $O(n\cdot|\Sigma|)$ 바이트임을 알 수 있다. 그러나 |Σ|가 작은 염기 서열이나 단백질 서열의 경우, 실제 사용하는 공간을 계산해 보면 다른 알고리즘과 비슷하거나 더 적은 메모리 공간을 사용하게 된다. 위 알고리즘에서 추가적으로 필요한 메모리 공간은 비트 패턴, 서브테이블I,II와 Lookup 테이블을 위한 공간이며 각각의 사용 공간은 다음과 같다.

(비트 패턴의 메모리 공간)

= (알파벳 수) × (문자열의 길이)

$= |\Sigma|\cdot n$ (비트)

(서브테이블I의 메모리 공간)

= (알파벳 수) × (큰 구간의 개수) × (저장 값 하나의 공간)

$= (|\Sigma|)\times(n/\log^2 n)\times(\log n)$

$= |\Sigma|\cdot n/\log n$ (비트)

(서브테이블II의 메모리 공간)

= (알파벳 수) × (작은 구간의 개수) × (저장 값 하나의 공간)

$= (|\Sigma|)\times(n/\log n)\times(2\log\log n)$

$= 2|\Sigma|\cdot n\cdot\log\log n/\log n$ (비트)

(Lookup 테이블의 메모리 공간)

= (숫자의 개수) × (저장 값 하나의 공간)

$= (n^{1/2})\times(\log\log n^{1/2})$

$= n^{1/2}\cdot\log\log n^{1/2}$ (비트)

따라서, 큰 구간을 1024, 작은 구간을 32로 하고, 서브테이블I,II와 Lookup 테이블에서 저장 값 하나의 공간을 각각 4, 2, 1바이트로 하여 구현할 경우, 대략 $0.2\times|\Sigma|\cdot n$ 바이트의 공간을 차지하게 되며, [표1]에서와 같이 |Σ|의 크기에 따라 다른 알고리즘과 비슷하거나 적은 공간을 사용하게 됨을 알 수 있다.

[표1] 각 알고리즘의 메모리 사용 공간 비교

|Σ|	ESA*	XYZ**	Table***
4	2n	6n	0.8n
20	2n	6n	4n

* ESA : Abouelhoda-Ohlebusch-Kurtz 알고리즘[4]
** XYZ : Sim-Kim-Park-Park 알고리즘[5]
*** Table : 본 논문의 알고리즘

4. 실험

4.1. 실험 설정

실험은 알파벳 크기가 4인 경우와 20인 경우, 두 가지에 대하여 실행하였다. 실험을 위해 먼저 크기가 각각 1, 10, 20, 30 메가 바이트인 임의의 텍스트를 생성하였다. 각 텍스트에 대한 1,000,000개의 패턴을 그 텍스트로부터 임의로 추출하였고, 실제 패턴 검색에서 패턴을 찾지 못하는 경우가 자주 발생하는 것

을 고려하여 이 중 절반의 패턴은 앞뒤를 뒤집어서 사용하였으며, 패턴들의 길이는 10-20, 20-30, 30-40인 세 가지의 경우에 대해 실험하였다.

실험에 사용된 운영체제는 LINUX(RedHat 8.0)이고 프로세서는 Intel Pentium4 Xeon 2.4GHz Dual, 주 기억 장치는 2Gbyte인 하드웨어가 사용되었다. 시간 측정에 있어서 패턴 검색을 위해 필요한 추가적인 자료 구조를 만드는데 걸리는 시간은 제외되었고, 1,000,000개의 패턴을 한 번씩 검색하는데 걸리는 전체 시간을 측정하였다. 또한 패턴 검색의 시간은 패턴이 위치하는 텍스트의 인덱스, 또는 텍스트의 여러 인덱스들이 접미사 배열 상에 위치하는 인덱스의 범위를 반환하는 것까지의 시간을 의미하며, 각 위치를 모두 나열하는데 걸리는 시간은 계산되지 않았다. 최종적인 실험값은 10번의 실험을 수행하여 최고값과 최저값을 뺀 나머지 값들의 평균을 취하였다.

4.2. 실험 결과

[표2]에서 보듯이 $|\Sigma|=4$인 경우, 본 논문의 알고리즘이 다른 두 알고리즘보다 1.5배-2배 정도 빠르며, [표3]에서 보듯이 $|\Sigma|=20$인 경우, 역시 Sim-Kim-Park-Park 알고리즘보다는 1.5배 정도, Abouelhoda-Ohlebusch-Kurtz 알고리즘보다는 4배 정도 빠름을 알 수 있다. Abouelhoda-Ohlebusch-Kurtz 알고리즘의 경우, $|\Sigma|$가 커질 때, 상당히 느려지는 것을 볼 수 있는데, 이것은 최대 $O(|\Sigma| \cdot n)$의 수행 시간을 가지기 때문이다. 나머지 두 알고리즘은 $|\Sigma|=20$일 때 수행 시간이 줄어들었는데, 이것은 앞뒤를 뒤집은 패턴들을 검색할 때 더 이상 일치하는 부분이 없어 검색을 중단하게 되는 시점이 $|\Sigma|=4$일 때보다 더 빨리 나타나기 때문이다.

[표2] 검색 시간 ($|\Sigma|=4$인 경우, 단위:초)

텍스트	패턴길이:10-20			패턴길이:20-30			패턴길이:30-40		
	ESA	XYZ	Table	ESA	XYZ	Table	ESA	XYZ	Table
1M	4.50	5.62	2.73	4.69	5.66	2.64	4.72	5.78	2.80
10M	7.05	8.07	5.18	7.30	8.46	5.27	7.37	8.51	5.36
20M	8.16	8.52	5.28	8.30	9.11	5.85	8.22	8.96	5.86
30M	9.07	9.05	5.79	9.25	10.06	6.23	9.14	9.88	6.30

[표3] 검색 시간 ($|\Sigma|=20$인 경우, 단위:초)

텍스트	패턴길이:10-20			패턴길이:20-30			패턴길이:30-40		
	ESA	XYZ	Table	ESA	XYZ	Table	ESA	XYZ	Table
1M	7.90	3.16	2.05	7.91	3.21	2.06	7.98	3.23	2.11
10M	14.00	4.65	3.33	13.99	4.76	3.40	14.00	4.77	3.36
20M	16.04	5.24	3.65	15.81	5.27	3.61	15.83	5.26	3.65
30M	17.01	5.75	4.00	17.03	5.60	4.01	17.04	5.65	3.95

5. 결론

본 논문에서는 $0.2\times|\Sigma|\cdot n$의 공간을 사용하여 $O(m)$ 시간에 패턴을 검색할 수 있는 알고리즘을 제시하였다. 실제 실험 결과에서도 $O(m\cdot|\Sigma|)$ 또는 $O(m\cdot\log|\Sigma|)$ 시간 알고리즘보다 빠르게 수행되는 것을 보였으며, 특히 $|\Sigma|$가 작은 염기 서열이나 단백질 서열에서의 패턴 검색을 수행할 경우, 메모리 공간도 적게 사용함을 알 수 있다.

참고 문헌

[1] Udi Manber, Gene Myers, "Suffix array: A new method for on-line string searches", In Proc. of the 1st ACM-SIAM Symposium on Discrete Algorithms, 319-327, 1990.

[2] E. McCreight, "A space-economical suffix tree construction algorithm", Journal of the ACM, 23(2), 262-272, 1976.

[3] E. Ukkonen, "On-Line Construction of Suffix Trees", Algorithmica, 14(3), 249-260, 1995.

[4] M. Abouelhoda, E. Ohlebusch, and S. Kurtz, "Optimal exact string matching based on suffix arrays", In Proc. of the 9th International Symposium on String Processing and Information Retrieval, LNCS 2476, 31-43, 2002.

[5] J. Sim, D. Kim, H. Park and K. Park, "Linear-Time Search in Suffix Arrays", In Proc. of the 14th Australasian Workshop on Combinatorial Algorithms, 139-146, 2003.

[6] P. Ferragina and G. Manzini, "Opportunistic data structures with applications", IEEE Symposium on Foundation of Computer Science", 390-398, 2000.